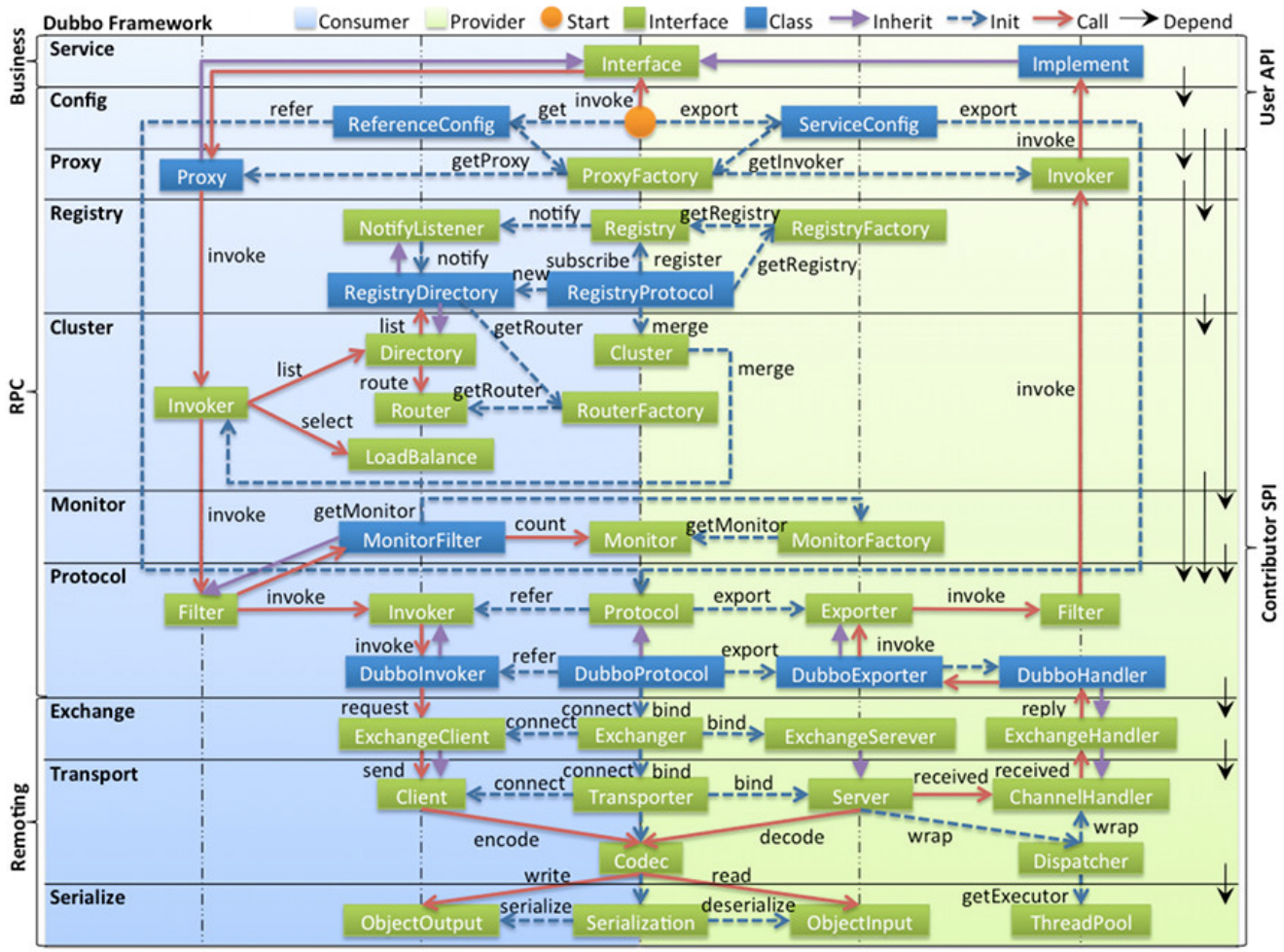
注:

1.阅读此篇之前你需要会dubbo的基本使用

2.需要知道netty的一些知识

3.此篇的源码使用版本为2.6.2

```
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>dubbo</artifactId>
    <version>2.6.2</version>
</dependency>
```



注:此图为dubbo官方给的结构图

1.dubbo原理 -启动解析、加载配置信息

1.测试代码以及配置文件

```
<!-- 服务提供者的配置文件 -->
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
```

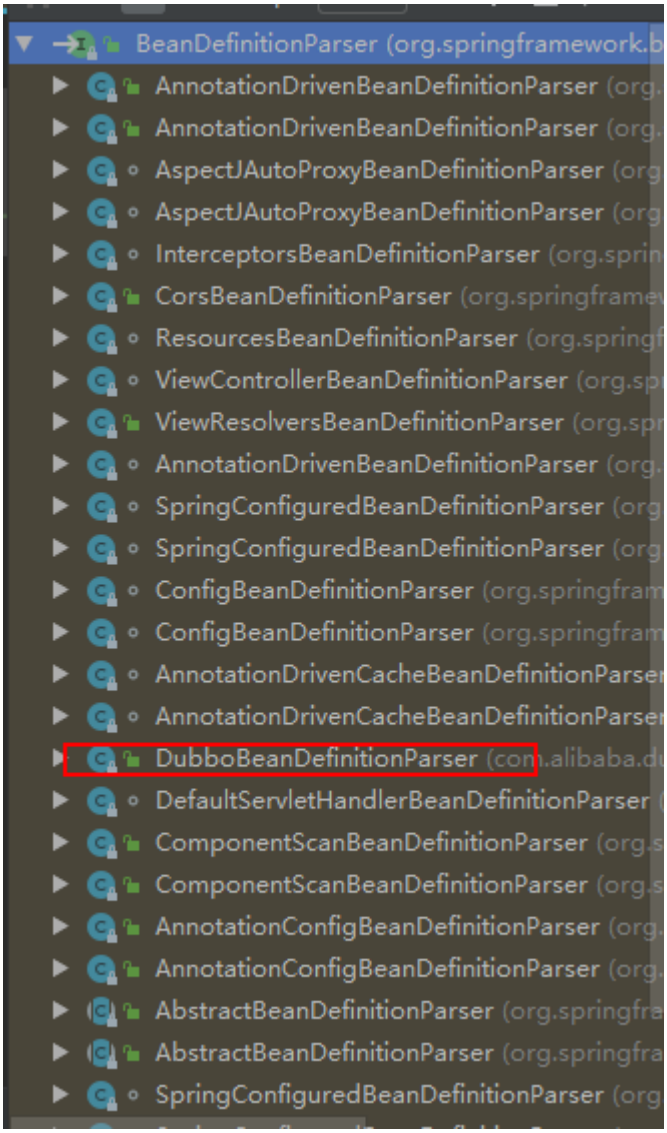
```
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://dubbo.apache.org/schema/dubbo http://dubbo.apache.org/schema/dubbo/dubbo.xsd
        http://code.alibabatech.com/schema/dubbo
http://code.alibabatech.com/schema/dubbo/dubbo.xsd">

    <!-- 提供方应用信息，用于计算依赖关系 -->
    <dubbo:application name="user-service-provider"></dubbo:application>

    <!-- 指定注册中心位置 -->
    <dubbo:registry protocol="zookeeper" address="192.168.1.102:2181" />

    <!-- 指定通信规则 通信协议 通信规则-->
    <dubbo:protocol name="dubbo" port="20880"></dubbo:protocol>

    <!-- 暴露服务 -->
    <dubbo:service interface="com.mgw.gmall.service.UserService" ref="userServiceImpl">
</dubbo:service>

    <bean id="userServiceImpl" class="com.mgw.gmall.service.impl.UserServiceImpl"></bean>

    <!-- 连接监控中心 -->
    <dubbo:monitor protocol="registry"></dubbo:monitor>

</beans>
```

```
//测试的对外暴露服务提供者
public class UserServiceImpl implements UserService {


    public List<UserAddress> getUserAddressList(String userId) {

        //TODO...

        return new ArrayList();
    }
}
//测试主类
public class MainApplication {

    public static void main(String[] args) throws IOException {

        ClassPathXmlApplicationContext ioc = new ClassPathXmlApplicationContext("provider.xml");

        ioc.start();

        System.out.println("start....");
        System.in.read();

    }
}
```

首选得知道一个接口BeanDefinitionParser这个，这是spring提供的一个接口，用来做bean的解析的

```
public interface BeanDefinitionParser {

    /**
     * Parse the specified {@link Element} and register the resulting
     * {@link BeanDefinition BeanDefinition(s)} with the
     * {@link org.springframework.beans.factory.xml.ParserContext#getRegistry()
BeanDefinitionRegistry}
     * embedded in the supplied {@link ParserContext}.
     * <p>Implementations must return the primary {@link BeanDefinition} that results
     * from the parse if they will ever be used in a nested fashion (for example as
     * an inner tag in a {@code <property/>} tag). Implementations may return
     * {@code null} if they will <strong>not</strong> be used in a nested fashion.
     * @param element the element that is to be parsed into one or more {@link BeanDefinition
BeanDefinitions}
     * @param parserContext the object encapsulating the current state of the parsing process;
     * provides access to a {@link
org.springframework.beans.factory.support.BeanDefinitionRegistry}
     * @return the primary {@link BeanDefinition}
     */
    BeanDefinition parse(Element element, ParserContext parserContext);

}
```

看下面的这个接口的主要继承类



当你引入dubbo的jar包时，就会多出来一个DubboBeanDefinitionParser的dubbo相关的bean的解析类，这是整个dubbo配置bean的主要解析类

```
//这个类是通过名称空间来得到相关的bean处理器
public class DubboNamespaceHandler extends NamespaceHandlerSupport {

    static {
        Version.checkDuplicate(DubboNamespaceHandler.class);
    }

    //这个初始化函数很重要  因为它是用来解析dubbo的配置bean的，看下标签名称，是不是很熟悉？没错，就是dubbo的配置文
件
    @Override
    public void init() {
        registerBeanDefinitionParser("application", new
DubboBeanDefinitionParser(ApplicationConfig.class, true));
        registerBeanDefinitionParser("module", new DubboBeanDefinitionParser(ModuleConfig.class,
true));
        registerBeanDefinitionParser("registry", new
DubboBeanDefinitionParser(RegistryConfig.class, true));
```

```
        registerBeanDefinitionParser("monitor", new DubboBeanDefinitionParser(MonitorConfig.class,
true));
        registerBeanDefinitionParser("provider", new
DubboBeanDefinitionParser(ProviderConfig.class, true));
        registerBeanDefinitionParser("consumer", new
DubboBeanDefinitionParser(ConsumerConfig.class, true));
        registerBeanDefinitionParser("protocol", new
DubboBeanDefinitionParser(ProtocolConfig.class, true));
        registerBeanDefinitionParser("service", new DubboBeanDefinitionParser(ServiceBean.class,
true));
        registerBeanDefinitionParser("reference", new
DubboBeanDefinitionParser(ReferenceBean.class, false));
        registerBeanDefinitionParser("annotation", new AnnotationBeanDefinitionParser());
    }

}

//真正的dubbo解析类
public class DubboBeanDefinitionParser implements BeanDefinitionParser {

    private static final Logger logger = LoggerFactory.getLogger(DubboBeanDefinitionParser.class);
    private static final Pattern GROUP_AND_VERION = Pattern.compile("^[\\-.0-9_a-zA-Z]+(\\:[\\-.0-
9_a-zA-Z]+)?$");
    private final Class<?> beanClass;
    private final boolean required;

    public DubboBeanDefinitionParser(Class<?> beanClass, boolean required) {
        this.beanClass = beanClass;
        this.required = required;
    }

    //真正的解析方法
    @SuppressWarnings("unchecked")
    private static BeanDefinition parse(Element element, ParserContext parserContext, Class<?>
beanClass, boolean required) {
        //创建一个beanDefinition类  这个类就是spring中用来保存bean定义信息的类  是通用的
        RootBeanDefinition beanDefinition = new RootBeanDefinition();
        beanDefinition.setBeanClass(beanClass);
        beanDefinition.setLazyInit(false);
        //拿到标签的ID
        String id = element.getAttribute("id");
        if ((id == null || id.length() == 0) && required) {
            //拿到标签的name的值
            /*
            举个例子:<dubbo:application name="user-service-provider"></dubbo:application>
            如果是这个标签，此时拿到的就是user-service-provider 就是这个应用的唯一标识  因为dubbo中规定应用名
称是唯一的
            */
            String generatedBeanName = element.getAttribute("name");
            if (generatedBeanName == null || generatedBeanName.length() == 0) {
                //如果拿不到 判断标签是否是protocol协议标签  如果是给默认值
                //<dubbo:protocol name="dubbo" port="20880"></dubbo:protocol>
                if (ProtocolConfig.class.equals(beanClass)) {
                    generatedBeanName = "dubbo";
                } else {
                    //否则则去拿需要暴露的服务的标签
```

```
                    //<dubbo:service interface="com.mgw.gmall.service.UserService"
ref="userServiceImpl"></dubbo:service>
                    generatedBeanName = element.getAttribute("interface");
                }
            }
            if (generatedBeanName == null || generatedBeanName.length() == 0) {
                generatedBeanName = beanClass.getName();
            }
            id = generatedBeanName;
            int counter = 2;
            while (parserContext.getRegistry().containsBeanDefinition(id)) {
                id = generatedBeanName + (counter++);
            }
        }
        if (id != null && id.length() > 0) {
            //防止重复定义
            if (parserContext.getRegistry().containsBeanDefinition(id)) {
                throw new IllegalStateException("Duplicate spring bean id " + id);
            }
            //注册bean
            parserContext.getRegistry().registerBeanDefinition(id, beanDefinition);
            beanDefinition.getPropertyValues().addPropertyValue("id", id);
        }
        //下面开始针对每一种不同的标签  做相应的值的封装
        //针对protocol协议这种标签做特点的值的设置
        //示例:<dubbo:protocol name="dubbo" port="20880"></dubbo:protocol>
        if (ProtocolConfig.class.equals(beanClass)) {
            //依次循环注册中心已经注册了的bean信息
            for (String name : parserContext.getRegistry().getBeanDefinitionNames()) {
                BeanDefinition definition = parserContext.getRegistry().getBeanDefinition(name);
                //找到其中那个注册了属性为protocol的那条的注册信息
                PropertyValue property =
definition.getPropertyValues().getPropertyValue("protocol");
                if (property != null) {
                    Object value = property.getValue();
                    if (value instanceof ProtocolConfig && id.equals(((ProtocolConfig)
value).getName())) {
                        definition.getPropertyValues().addPropertyValue("protocol", new
RuntimeBeanReference(id));
                    }
                }
            }
        } else if (ServiceBean.class.equals(beanClass)) {
            String className = element.getAttribute("class");
            if (className != null && className.length() > 0) {
                RootBeanDefinition classDefinition = new RootBeanDefinition();
                classDefinition.setBeanClass(ReflectUtils.forName(className));
                classDefinition.setLazyInit(false);
                parseProperties(element.getChildNodes(), classDefinition);
                beanDefinition.getPropertyValues().addPropertyValue("ref", new
BeanDefinitionHolder(classDefinition, id + "Impl"));
            }
        } else if (ProviderConfig.class.equals(beanClass)) {
            parseNested(element, parserContext, ServiceBean.class, true, "service", "provider",
id, beanDefinition);
        } else if (ConsumerConfig.class.equals(beanClass)) {
```

```
            parseNested(element, parserContext, ReferenceBean.class, false, "reference",
"consumer", id, beanDefinition);
        }
        Set<String> props = new HashSet<String>();
        ManagedMap parameters = null;
        for (Method setter : beanClass.getMethods()) {
            String name = setter.getName();
            if (name.length() > 3 && name.startsWith("set")
                    && Modifier.isPublic(setter.getModifiers())
                    && setter.getParameterTypes().length == 1) {
                Class<?> type = setter.getParameterTypes()[0];
                String property = StringUtils.camelToSplitName(name.substring(3, 4).toLowerCase()
+ name.substring(4), "-");
                props.add(property);
                Method getter = null;
                try {
                    getter = beanClass.getMethod("get" + name.substring(3), new Class<?>[0]);
                } catch (NoSuchMethodException e) {
                    try {
                        getter = beanClass.getMethod("is" + name.substring(3), new Class<?>[0]);
                    } catch (NoSuchMethodException e2) {
                    }
                }
                if (getter == null
                        || !Modifier.isPublic(getter.getModifiers())
                        || !type.equals(getter.getReturnType())) {
                    continue;
                }
                if ("parameters".equals(property)) {
                    parameters = parseParameters(element.getChildNodes(), beanDefinition);
                } else if ("methods".equals(property)) {
                    parseMethods(id, element.getChildNodes(), beanDefinition, parserContext);
                } else if ("arguments".equals(property)) {
                    parseArguments(id, element.getChildNodes(), beanDefinition, parserContext);
                } else {
                    String value = element.getAttribute(property);
                    if (value != null) {
                        value = value.trim();
                        if (value.length() > 0) {
                            if ("registry".equals(property) &&
RegistryConfig.NO_AVAILABLE.equalsIgnoreCase(value)) {
                                RegistryConfig registryConfig = new RegistryConfig();
                                registryConfig.setAddress(RegistryConfig.NO_AVAILABLE);
                                beanDefinition.getPropertyValues().addPropertyValue(property,
registryConfig);
                            } else if ("registry".equals(property) && value.indexOf(',') != -1) {
                                parseMultiRef("registries", value, beanDefinition, parserContext);
                            } else if ("provider".equals(property) && value.indexOf(',') != -1) {
                                parseMultiRef("providers", value, beanDefinition, parserContext);
                            } else if ("protocol".equals(property) && value.indexOf(',') != -1) {
                                parseMultiRef("protocols", value, beanDefinition, parserContext);
                            } else {
                                Object reference;
                                if (isPrimitive(type)) {
                                    if ("async".equals(property) && "false".equals(value)
                                            || "timeout".equals(property) && "0".equals(value)
                                            || "delay".equals(property) && "0".equals(value)
```

```
                                        || "version".equals(property) && "0.0.0".equals(value)
                                        || "stat".equals(property) && "-1".equals(value)
                                        || "reliable".equals(property) &&
"false".equals(value)) {
                                    // backward compatibility for the default value in old
version's xsd
                                    value = null;
                                }
                                reference = value;
                            } else if ("protocol".equals(property)
                                    &&
ExtensionLoader.getExtensionLoader(Protocol.class).hasExtension(value)
                                    &&
(!parserContext.getRegistry().containsBeanDefinition(value)
                                    ||
!ProtocolConfig.class.getName().equals(parserContext.getRegistry().getBeanDefinition(value).getBea
nClassName()))) {
                                if ("dubbo:provider".equals(element.getTagName())) {
                                    logger.warn("Recommended replace <dubbo:provider
protocol=\"" + value + "\" ... /> to <dubbo:protocol name=\"" + value + "\" ... />");
                                }
                                // backward compatibility
                                ProtocolConfig protocol = new ProtocolConfig();
                                protocol.setName(value);
                                reference = protocol;
                            } else if ("onreturn".equals(property)) {
                                int index = value.lastIndexOf(".");
                                String returnRef = value.substring(0, index);
                                String returnMethod = value.substring(index + 1);
                                reference = new RuntimeBeanReference(returnRef);

 beanDefinition.getPropertyValues().addPropertyValue("onreturnMethod", returnMethod);
                            } else if ("onthrow".equals(property)) {
                                int index = value.lastIndexOf(".");
                                String throwRef = value.substring(0, index);
                                String throwMethod = value.substring(index + 1);
                                reference = new RuntimeBeanReference(throwRef);

 beanDefinition.getPropertyValues().addPropertyValue("onthrowMethod", throwMethod);
                            } else if ("oninvoke".equals(property)) {
                                int index = value.lastIndexOf(".");
                                String invokeRef = value.substring(0, index);
                                String invokeRefMethod = value.substring(index + 1);
                                reference = new RuntimeBeanReference(invokeRef);

 beanDefinition.getPropertyValues().addPropertyValue("oninvokeMethod", invokeRefMethod);
                            }else {
                                if ("ref".equals(property) &&
parserContext.getRegistry().containsBeanDefinition(value)) {
                                    BeanDefinition refBean =
parserContext.getRegistry().getBeanDefinition(value);
                                    if (!refBean.isSingleton()) {
                                        throw new IllegalStateException("The exported service
ref " + value + " must be singleton! Please set the " + value + " bean scope to singleton, eg:
<bean id=\"" + value + "\" scope=\"singleton\" ...>");
                                    }
                                }
```

```
                                    reference = new RuntimeBeanReference(value);
                                }
                                beanDefinition.getPropertyValues().addPropertyValue(property,
reference);
                            }
                        }
                    }
                }
            }
        }
        NamedNodeMap attributes = element.getAttributes();
        int len = attributes.getLength();
        for (int i = 0; i < len; i++) {
            Node node = attributes.item(i);
            String name = node.getLocalName();
            if (!props.contains(name)) {
                if (parameters == null) {
                    parameters = new ManagedMap();
                }
                String value = node.getNodeValue();
                parameters.put(name, new TypedStringValue(value, String.class));
            }
        }
        if (parameters != null) {
            beanDefinition.getPropertyValues().addPropertyValue("parameters", parameters);
        }
        return beanDefinition;
    }

    private static boolean isPrimitive(Class<?> cls) {
        return cls.isPrimitive() || cls == Boolean.class || cls == Byte.class
                || cls == Character.class || cls == Short.class || cls == Integer.class
                || cls == Long.class || cls == Float.class || cls == Double.class
                || cls == String.class || cls == Date.class || cls == Class.class;
    }

    @SuppressWarnings("unchecked")
    private static void parseMultiRef(String property, String value, RootBeanDefinition
beanDefinition,
                                      ParserContext parserContext) {
        String[] values = value.split("\\s*[,]+\\s*");
        ManagedList list = null;
        for (int i = 0; i < values.length; i++) {
            String v = values[i];
            if (v != null && v.length() > 0) {
                if (list == null) {
                    list = new ManagedList();
                }
                list.add(new RuntimeBeanReference(v));
            }
        }
        beanDefinition.getPropertyValues().addPropertyValue(property, list);
    }

    private static void parseNested(Element element, ParserContext parserContext, Class<?>
beanClass, boolean required, String tag, String property, String ref, BeanDefinition
beanDefinition) {
```

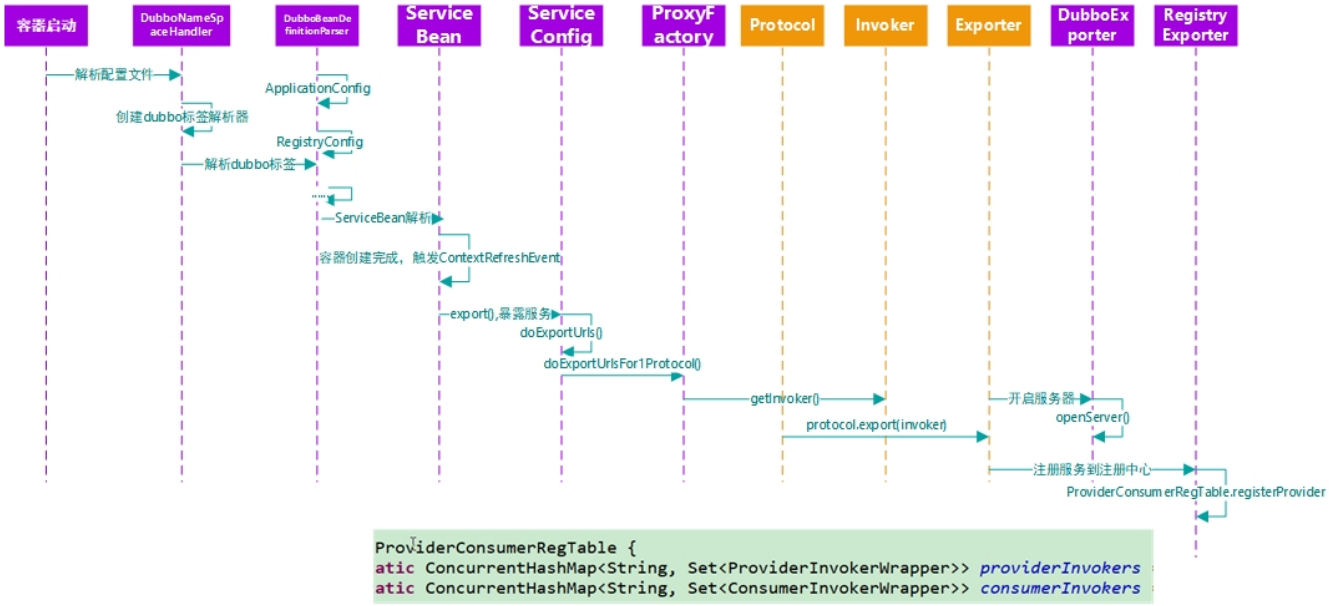
```
        NodeList nodeList = element.getChildNodes();
        if (nodeList != null && nodeList.getLength() > 0) {
            boolean first = true;
            for (int i = 0; i < nodeList.getLength(); i++) {
                Node node = nodeList.item(i);
                if (node instanceof Element) {
                    if (tag.equals(node.getNodeName())
                            || tag.equals(node.getLocalName())) {
                        if (first) {
                            first = false;
                            String isDefault = element.getAttribute("default");
                            if (isDefault == null || isDefault.length() == 0) {
                                beanDefinition.getPropertyValues().addPropertyValue("default",
"false");
                            }
                        }
                        BeanDefinition subDefinition = parse((Element) node, parserContext,
beanClass, required);
                        if (subDefinition != null && ref != null && ref.length() > 0) {
                            subDefinition.getPropertyValues().addPropertyValue(property, new
RuntimeBeanReference(ref));
                        }
                    }
                }
            }
        }
    }

    private static void parseProperties(NodeList nodeList, RootBeanDefinition beanDefinition) {
        if (nodeList != null && nodeList.getLength() > 0) {
            for (int i = 0; i < nodeList.getLength(); i++) {
                Node node = nodeList.item(i);
                if (node instanceof Element) {
                    if ("property".equals(node.getNodeName())
                            || "property".equals(node.getLocalName())) {
                        String name = ((Element) node).getAttribute("name");
                        if (name != null && name.length() > 0) {
                            String value = ((Element) node).getAttribute("value");
                            String ref = ((Element) node).getAttribute("ref");
                            if (value != null && value.length() > 0) {
                                beanDefinition.getPropertyValues().addPropertyValue(name, value);
                            } else if (ref != null && ref.length() > 0) {
                                beanDefinition.getPropertyValues().addPropertyValue(name, new
RuntimeBeanReference(ref));
                            } else {
                                throw new UnsupportedOperationException("Unsupported <property
name=\"" + name + "\"> sub tag, Only supported <property name=\"" + name + "\" ref=\"...\" /> or
<property name=\"" + name + "\" value=\"...\" />");
                            }
                        }
                    }
                }
            }
        }
    }

    @SuppressWarnings("unchecked")
```

```
    private static ManagedMap parseParameters(NodeList nodeList, RootBeanDefinition
beanDefinition) {
        if (nodeList != null && nodeList.getLength() > 0) {
            ManagedMap parameters = null;
            for (int i = 0; i < nodeList.getLength(); i++) {
                Node node = nodeList.item(i);
                if (node instanceof Element) {
                    if ("parameter".equals(node.getNodeName())
                            || "parameter".equals(node.getLocalName())) {
                        if (parameters == null) {
                            parameters = new ManagedMap();
                        }
                        String key = ((Element) node).getAttribute("key");
                        String value = ((Element) node).getAttribute("value");
                        boolean hide = "true".equals(((Element) node).getAttribute("hide"));
                        if (hide) {
                            key = Constants.HIDE_KEY_PREFIX + key;
                        }
                        parameters.put(key, new TypedStringValue(value, String.class));
                    }
                }
            }
            return parameters;
        }
        return null;
    }

    @SuppressWarnings("unchecked")
    private static void parseMethods(String id, NodeList nodeList, RootBeanDefinition
beanDefinition,
                                     ParserContext parserContext) {
        if (nodeList != null && nodeList.getLength() > 0) {
            ManagedList methods = null;
            for (int i = 0; i < nodeList.getLength(); i++) {
                Node node = nodeList.item(i);
                if (node instanceof Element) {
                    Element element = (Element) node;
                    if ("method".equals(node.getNodeName()) ||
"method".equals(node.getLocalName())) {
                        String methodName = element.getAttribute("name");
                        if (methodName == null || methodName.length() == 0) {
                            throw new IllegalStateException("<dubbo:method> name attribute ==
null");
                        }
                        if (methods == null) {
                            methods = new ManagedList();
                        }
                        BeanDefinition methodBeanDefinition = parse(((Element) node),
                                parserContext, MethodConfig.class, false);
                        String name = id + "." + methodName;
                        BeanDefinitionHolder methodBeanDefinitionHolder = new
BeanDefinitionHolder(
                                methodBeanDefinition, name);
                        methods.add(methodBeanDefinitionHolder);
                    }
                }
            }
```

```
            if (methods != null) {
                beanDefinition.getPropertyValues().addPropertyValue("methods", methods);
            }
        }
    }

    @SuppressWarnings("unchecked")
    private static void parseArguments(String id, NodeList nodeList, RootBeanDefinition
beanDefinition,
                                       ParserContext parserContext) {
        if (nodeList != null && nodeList.getLength() > 0) {
            ManagedList arguments = null;
            for (int i = 0; i < nodeList.getLength(); i++) {
                Node node = nodeList.item(i);
                if (node instanceof Element) {
                    Element element = (Element) node;
                    if ("argument".equals(node.getNodeName()) ||
"argument".equals(node.getLocalName())) {
                        String argumentIndex = element.getAttribute("index");
                        if (arguments == null) {
                            arguments = new ManagedList();
                        }
                        BeanDefinition argumentBeanDefinition = parse(((Element) node),
                                parserContext, ArgumentConfig.class, false);
                        String name = id + "." + argumentIndex;
                        BeanDefinitionHolder argumentBeanDefinitionHolder = new
BeanDefinitionHolder(
                                argumentBeanDefinition, name);
                        arguments.add(argumentBeanDefinitionHolder);
                    }
                }
            }
            if (arguments != null) {
                beanDefinition.getPropertyValues().addPropertyValue("arguments", arguments);
            }
        }
    }

    @Override
    public BeanDefinition parse(Element element, ParserContext parserContext) {
        return parse(element, parserContext, beanClass, required);
    }

}
```

2.dubbo原理 -服务暴露

运行时的时序图



前面说过了初始化过程时会注册加载配置文件的过程，其中有一个

```
registerBeanDefinitionParser("service", new DubboBeanDefinitionParser(ServiceBean.class, true));
```

这个就是服务暴露主要的过程

```
public class ServiceBean<T> extends ServiceConfig<T> implements InitializingBean, DisposableBean,
ApplicationContextAware, ApplicationListener<ContextRefreshedEvent>, BeanNameAware {

    private static final long serialVersionUID = 213195494150089726L;

    private static transient ApplicationContext SPRING_CONTEXT;

    private final transient Service service;

    private transient ApplicationContext applicationContext;

    private transient String beanName;

    private transient boolean supportedApplicationListener;

    public ServiceBean() {
        super();
        this.service = null;
    }

    public ServiceBean(Service service) {
        super(service);
        this.service = service;
    }

    public static ApplicationContext getSpringContext() {
        return SPRING_CONTEXT;
    }

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) {
```

```
        this.applicationContext = applicationContext;
        SpringExtensionFactory.addApplicationContext(applicationContext);
        if (applicationContext != null) {
            SPRING_CONTEXT = applicationContext;
            try {
                Method method = applicationContext.getClass().getMethod("addApplicationListener",
new Class<?>[]{ApplicationListener.class}); // backward compatibility to spring 2.0.1
                method.invoke(applicationContext, new Object[]{this});
                supportedApplicationListener = true;
            } catch (Throwable t) {
                if (applicationContext instanceof AbstractApplicationContext) {
                    try {
                        Method method =
AbstractApplicationContext.class.getDeclaredMethod("addListener", new Class<?>[]
{ApplicationListener.class}); // backward compatibility to spring 2.0.1
                        if (!method.isAccessible()) {
                            method.setAccessible(true);
                        }
                        method.invoke(applicationContext, new Object[]{this});
                        supportedApplicationListener = true;
                    } catch (Throwable t2) {
                    }
                }
            }
        }
    }

    @Override
    public void setBeanName(String name) {
        this.beanName = name;
    }

    /**
     * Gets associated {@link Service}
     *
     * @return associated {@link Service}
     */
    public Service getService() {
        return service;
    }

    //ApplicationListener接口的方法 当ioc容器刷新完毕后回调此方法
    @Override
    public void onApplicationEvent(ContextRefreshedEvent event) {
        if (isDelay() && !isExported() && !isUnexported()) {
            if (logger.isInfoEnabled()) {
                logger.info("The service ready on spring started. service: " + getInterface());
            }
            //暴露服务
            export();
        }
    }

    private boolean isDelay() {
        Integer delay = getDelay();
        ProviderConfig provider = getProvider();
        if (delay == null && provider != null) {
```

```
            delay = provider.getDelay();
        }
        return supportedApplicationListener && (delay == null || delay == -1);
    }

    //InitializingBean接口的方法 在对象属性设置完成后回调此方法
    @Override
    @SuppressWarnings({"unchecked", "deprecation"})
    public void afterPropertiesSet() throws Exception {
        //做provider标签内容的保存
        if (getProvider() == null) {
            Map<String, ProviderConfig> providerConfigMap = applicationContext == null ? null :
BeanFactoryUtils.beansOfTypeIncludingAncestors(applicationContext, ProviderConfig.class, false,
false);
            if (providerConfigMap != null && providerConfigMap.size() > 0) {
                Map<String, ProtocolConfig> protocolConfigMap = applicationContext == null ? null
: BeanFactoryUtils.beansOfTypeIncludingAncestors(applicationContext, ProtocolConfig.class, false,
false);
                if ((protocolConfigMap == null || protocolConfigMap.size() == 0)
                        && providerConfigMap.size() > 1) { // backward compatibility
                    List<ProviderConfig> providerConfigs = new ArrayList<ProviderConfig>();
                    for (ProviderConfig config : providerConfigMap.values()) {
                        if (config.isDefault() != null && config.isDefault().booleanValue()) {
                            providerConfigs.add(config);
                        }
                    }
                    if (!providerConfigs.isEmpty()) {
                        setProviders(providerConfigs);
                    }
                } else {
                    ProviderConfig providerConfig = null;
                    for (ProviderConfig config : providerConfigMap.values()) {
                        if (config.isDefault() == null || config.isDefault().booleanValue()) {
                            if (providerConfig != null) {
                                throw new IllegalStateException("Duplicate provider configs: " +
providerConfig + " and " + config);
                            }
                            providerConfig = config;
                        }
                    }
                    if (providerConfig != null) {
                        setProvider(providerConfig);
                    }
                }
            }
        }
        //做application标签内容的保存
        if (getApplication() == null
                && (getProvider() == null || getProvider().getApplication() == null)) {
            Map<String, ApplicationConfig> applicationConfigMap = applicationContext == null ?
null : BeanFactoryUtils.beansOfTypeIncludingAncestors(applicationContext, ApplicationConfig.class,
false, false);
            if (applicationConfigMap != null && applicationConfigMap.size() > 0) {
                ApplicationConfig applicationConfig = null;
                for (ApplicationConfig config : applicationConfigMap.values()) {
                    if (config.isDefault() == null || config.isDefault().booleanValue()) {
                        if (applicationConfig != null) {
```

```
                            throw new IllegalStateException("Duplicate application configs: " +
applicationConfig + " and " + config);
                        }
                        applicationConfig = config;
                    }
                }
                if (applicationConfig != null) {
                    setApplication(applicationConfig);
                }
            }
        }
        //做module内容的保存
        if (getModule() == null
                && (getProvider() == null || getProvider().getModule() == null)) {
            Map<String, ModuleConfig> moduleConfigMap = applicationContext == null ? null :
BeanFactoryUtils.beansOfTypeIncludingAncestors(applicationContext, ModuleConfig.class, false,
false);
            if (moduleConfigMap != null && moduleConfigMap.size() > 0) {
                ModuleConfig moduleConfig = null;
                for (ModuleConfig config : moduleConfigMap.values()) {
                    if (config.isDefault() == null || config.isDefault().booleanValue()) {
                        if (moduleConfig != null) {
                            throw new IllegalStateException("Duplicate module configs: " +
moduleConfig + " and " + config);
                        }
                        moduleConfig = config;
                    }
                }
                if (moduleConfig != null) {
                    setModule(moduleConfig);
                }
            }
        }
         //做registry内容的保存
        if ((getRegistries() == null || getRegistries().isEmpty())
                && (getProvider() == null || getProvider().getRegistries() == null ||
getProvider().getRegistries().isEmpty())
                && (getApplication() == null || getApplication().getRegistries() == null ||
getApplication().getRegistries().isEmpty())) {
            Map<String, RegistryConfig> registryConfigMap = applicationContext == null ? null :
BeanFactoryUtils.beansOfTypeIncludingAncestors(applicationContext, RegistryConfig.class, false,
false);
            if (registryConfigMap != null && registryConfigMap.size() > 0) {
                List<RegistryConfig> registryConfigs = new ArrayList<RegistryConfig>();
                for (RegistryConfig config : registryConfigMap.values()) {
                    if (config.isDefault() == null || config.isDefault().booleanValue()) {
                        registryConfigs.add(config);
                    }
                }
                if (registryConfigs != null && !registryConfigs.isEmpty()) {
                    super.setRegistries(registryConfigs);
                }
            }
        }
        //做monitor内容的保存
        if (getMonitor() == null
                && (getProvider() == null || getProvider().getMonitor() == null)
```

```
                && (getApplication() == null || getApplication().getMonitor() == null)) {
            Map<String, MonitorConfig> monitorConfigMap = applicationContext == null ? null :
BeanFactoryUtils.beansOfTypeIncludingAncestors(applicationContext, MonitorConfig.class, false,
false);
            if (monitorConfigMap != null && monitorConfigMap.size() > 0) {
                MonitorConfig monitorConfig = null;
                for (MonitorConfig config : monitorConfigMap.values()) {
                    if (config.isDefault() == null || config.isDefault().booleanValue()) {
                        if (monitorConfig != null) {
                            throw new IllegalStateException("Duplicate monitor configs: " +
monitorConfig + " and " + config);
                        }
                        monitorConfig = config;
                    }
                }
                if (monitorConfig != null) {
                    setMonitor(monitorConfig);
                }
            }
        }
        //做protocol内容的保存
        if ((getProtocols() == null || getProtocols().isEmpty())
                && (getProvider() == null || getProvider().getProtocols() == null ||
getProvider().getProtocols().isEmpty())) {
            Map<String, ProtocolConfig> protocolConfigMap = applicationContext == null ? null :
BeanFactoryUtils.beansOfTypeIncludingAncestors(applicationContext, ProtocolConfig.class, false,
false);
            if (protocolConfigMap != null && protocolConfigMap.size() > 0) {
                List<ProtocolConfig> protocolConfigs = new ArrayList<ProtocolConfig>();
                for (ProtocolConfig config : protocolConfigMap.values()) {
                    if (config.isDefault() == null || config.isDefault().booleanValue()) {
                        protocolConfigs.add(config);
                    }
                }
                if (protocolConfigs != null && !protocolConfigs.isEmpty()) {
                    super.setProtocols(protocolConfigs);
                }
            }
        }
        if (getPath() == null || getPath().length() == 0) {
            if (beanName != null && beanName.length() > 0
                    && getInterface() != null && getInterface().length() > 0
                    && beanName.startsWith(getInterface())) {
                setPath(beanName);
            }
        }
        if (!isDelay()) {
            export();
        }
    }

    @Override
    public void destroy() throws Exception {
        // This will only be called for singleton scope bean, and expected to be called by spring
shutdown hook when BeanFactory/ApplicationContext destroys.
        // We will guarantee dubbo related resources being released with dubbo shutdown hook.
        //unexport();
```

```
    }

    // merged from dubbox
    @Override
    protected Class getServiceClass(T ref) {
        if (AopUtils.isAopProxy(ref)) {
            return AopUtils.getTargetClass(ref);
        }
        return super.getServiceClass(ref);
    }
}

//暴露服务
public synchronized void export() {
    //做些判断之类的
    if (provider != null) {
        if (export == null) {
            export = provider.getExport();
        }
        if (delay == null) {
            delay = provider.getDelay();
        }
    }
    if (export != null && !export) {
        return;
    }

    if (delay != null && delay > 0) {
        delayExportExecutor.schedule(new Runnable() {
            @Override
            public void run() {
                doExport();
            }
        }, delay, TimeUnit.MILLISECONDS);
    } else {
        //暴露
        doExport();
    }
}

//暴露
protected synchronized void doExport() {
    //前面就还是相关信息的获取等
    if (unexported) {
        throw new IllegalStateException("Already unexported!");
    }
    if (exported) {
        return;
    }
    exported = true;
    if (interfaceName == null || interfaceName.length() == 0) {
        throw new IllegalStateException("<dubbo:service interface=\"\" /> interface not allow
null!");
    }
    checkDefault();
    if (provider != null) {
        if (application == null) {
```

```
            application = provider.getApplication();
        }
        if (module == null) {
            module = provider.getModule();
        }
        if (registries == null) {
            registries = provider.getRegistries();
        }
        if (monitor == null) {
            monitor = provider.getMonitor();
        }
        if (protocols == null) {
            protocols = provider.getProtocols();
        }
    }
    if (module != null) {
        if (registries == null) {
            registries = module.getRegistries();
        }
        if (monitor == null) {
            monitor = module.getMonitor();
        }
    }
    if (application != null) {
        if (registries == null) {
            registries = application.getRegistries();
        }
        if (monitor == null) {
            monitor = application.getMonitor();
        }
    }
    if (ref instanceof GenericService) {
        interfaceClass = GenericService.class;
        if (StringUtils.isEmpty(generic)) {
            generic = Boolean.TRUE.toString();
        }
    } else {
        try {
            interfaceClass = Class.forName(interfaceName, true, Thread.currentThread()
                    .getContextClassLoader());
        } catch (ClassNotFoundException e) {
            throw new IllegalStateException(e.getMessage(), e);
        }
        checkInterfaceAndMethods(interfaceClass, methods);
        checkRef();
        generic = Boolean.FALSE.toString();
    }
    if (local != null) {
        if ("true".equals(local)) {
            local = interfaceName + "Local";
        }
        Class<?> localClass;
        try {
            localClass = ClassHelper.forNameWithThreadContextClassLoader(local);
        } catch (ClassNotFoundException e) {
            throw new IllegalStateException(e.getMessage(), e);
        }
```

```
        if (!interfaceClass.isAssignableFrom(localClass)) {
            throw new IllegalStateException("The local implementation class " +
localClass.getName() + " not implement interface " + interfaceName);
        }
    }
    if (stub != null) {
        if ("true".equals(stub)) {
            stub = interfaceName + "Stub";
        }
        Class<?> stubClass;
        try {
            stubClass = ClassHelper.forNameWithThreadContextClassLoader(stub);
        } catch (ClassNotFoundException e) {
            throw new IllegalStateException(e.getMessage(), e);
        }
        if (!interfaceClass.isAssignableFrom(stubClass)) {
            throw new IllegalStateException("The stub implementation class " + stubClass.getName()
+ " not implement interface " + interfaceName);
        }
    }
    //检查相关需要的信息
    checkApplication();
    checkRegistry();
    checkProtocol();
    appendProperties(this);
    checkStubAndMock(interfaceClass);
    if (path == null || path.length() == 0) {
        path = interfaceName;
    }
    //重点方法  执行暴露url的地址
    doExportUrls();
    ProviderModel providerModel = new ProviderModel(getUniqueServiceName(), this, ref);
    ApplicationModel.initProviderModel(getUniqueServiceName(), providerModel);
}
//执行暴露url的地址
private void doExportUrls() {
    //加载注册中心的信息
    //得到如下的信息:
    //registry://192.168.1.102:2181/com.alibaba.dubbo.registry.RegistryService?application=user-
service-provider&dubbo=2.6.2&pid=17524&registry=zookeeper&timestamp=1555762836973
    List<URL> registryURLs = loadRegistries(true);
    //拿到当前的协议
    //<dubbo:protocol name="dubbo" port="20880"></dubbo:protocol>
    for (ProtocolConfig protocolConfig : protocols) {
        //以相关的协议暴露服务
        doExportUrlsFor1Protocol(protocolConfig, registryURLs);
    }
}

//加载注册中心的信息
//<dubbo:registry protocol="zookeeper" address="192.168.1.102:2181" />
protected List<URL> loadRegistries(boolean provider) {
    checkRegistry();
    List<URL> registryList = new ArrayList<URL>();
    if (registries != null && !registries.isEmpty()) {
        for (RegistryConfig config : registries) {
            String address = config.getAddress();
```

```
            if (address == null || address.length() == 0) {
                address = Constants.ANYHOST_VALUE;
            }
            //springboot中的配置文件的dubbo.registry.address的配置
            String sysaddress = System.getProperty("dubbo.registry.address");
            if (sysaddress != null && sysaddress.length() > 0) {
                address = sysaddress;
            }
            if (address != null && address.length() > 0
                    && !RegistryConfig.NO_AVAILABLE.equalsIgnoreCase(address)) {
                Map<String, String> map = new HashMap<String, String>();
                appendParameters(map, application);
                appendParameters(map, config);
                map.put("path", RegistryService.class.getName());
                map.put("dubbo", Version.getVersion());
                map.put(Constants.TIMESTAMP_KEY, String.valueOf(System.currentTimeMillis()));
                if (ConfigUtils.getPid() > 0) {
                    map.put(Constants.PID_KEY, String.valueOf(ConfigUtils.getPid()));
                }
                if (!map.containsKey("protocol")) {
                    if
(ExtensionLoader.getExtensionLoader(RegistryFactory.class).hasExtension("remote")) {
                        map.put("protocol", "remote");
                    } else {
                        map.put("protocol", "dubbo");
                    }
                }
                List<URL> urls = UrlUtils.parseURLs(address, map);
                for (URL url : urls) {
                    url = url.addParameter(Constants.REGISTRY_KEY, url.getProtocol());
                    url = url.setProtocol(Constants.REGISTRY_PROTOCOL);
                    if ((provider && url.getParameter(Constants.REGISTER_KEY, true))
                            || (!provider && url.getParameter(Constants.SUBSCRIBE_KEY, true))) {
                        registryList.add(url);
                    }
                }
            }
        }
    }
    return registryList;
}
//以相关的协议暴露服务
private void doExportUrlsFor1Protocol(ProtocolConfig protocolConfig, List<URL> registryURLs) {
    //拿到协议的名称  dubbo官方推荐为dubbo协议
    String name = protocolConfig.getName();
    if (name == null || name.length() == 0) {
        //默认也是dubbo协议
        name = "dubbo";
    }
    //设置一些信息
    Map<String, String> map = new HashMap<String, String>();
    //设置时那边的服务  为服务提供者的
    map.put(Constants.SIDE_KEY, Constants.PROVIDER_SIDE);
    //设置dubbo的版本
    map.put(Constants.DUBBO_VERSION_KEY, Version.getVersion());
    //开始的时间
    map.put(Constants.TIMESTAMP_KEY, String.valueOf(System.currentTimeMillis()));
```

```
    if (ConfigUtils.getPid() > 0) {
        //设置pid
        map.put(Constants.PID_KEY, String.valueOf(ConfigUtils.getPid()));
    }
    //下面的大部分都是在田间属性
    appendParameters(map, application);
    appendParameters(map, module);
    appendParameters(map, provider, Constants.DEFAULT_KEY);
    appendParameters(map, protocolConfig);
    //上面的四个属性应该可以一目了然  就说说这个this吧 是下面这个 其实就是相关需要暴露的服务
    /*
    <dubbo:service path="com.mgw.gmall.service.UserService"
ref="com.mgw.gmall.service.impl.UserServiceImpl@7f8a9499"
uniqueServiceName="com.mgw.gmall.service.UserService" unexported="false"
interface="com.mgw.gmall.service.UserService" exported="true" generic="false"
id="com.mgw.gmall.service.UserService" />
    */
    appendParameters(map, this);
    //下面大部分同样也是添加属性什么的
    if (methods != null && !methods.isEmpty()) {
        for (MethodConfig method : methods) {
            appendParameters(map, method, method.getName());
            String retryKey = method.getName() + ".retry";
            if (map.containsKey(retryKey)) {
                String retryValue = map.remove(retryKey);
                if ("false".equals(retryValue)) {
                    map.put(method.getName() + ".retries", "0");
                }
            }
            List<ArgumentConfig> arguments = method.getArguments();
            if (arguments != null && !arguments.isEmpty()) {
                for (ArgumentConfig argument : arguments) {
                    // convert argument type
                    if (argument.getType() != null && argument.getType().length() > 0) {
                        Method[] methods = interfaceClass.getMethods();
                        // visit all methods
                        if (methods != null && methods.length > 0) {
                            for (int i = 0; i < methods.length; i++) {
                                String methodName = methods[i].getName();
                                // target the method, and get its signature
                                if (methodName.equals(method.getName())) {
                                    Class<?>[] argtypes = methods[i].getParameterTypes();
                                    // one callback in the method
                                    if (argument.getIndex() != -1) {
                                        if
(argtypes[argument.getIndex()].getName().equals(argument.getType())) {
                                            appendParameters(map, argument, method.getName() + "."
+ argument.getIndex());
                                        } else {
                                            throw new IllegalArgumentException("argument config
error : the index attribute and type attribute not match :index :" + argument.getIndex() + ",
type:" + argument.getType());
                                        }
                                    } else {
                                        // multiple callbacks in the method
                                        for (int j = 0; j < argtypes.length; j++) {
                                            Class<?> argclazz = argtypes[j];
```

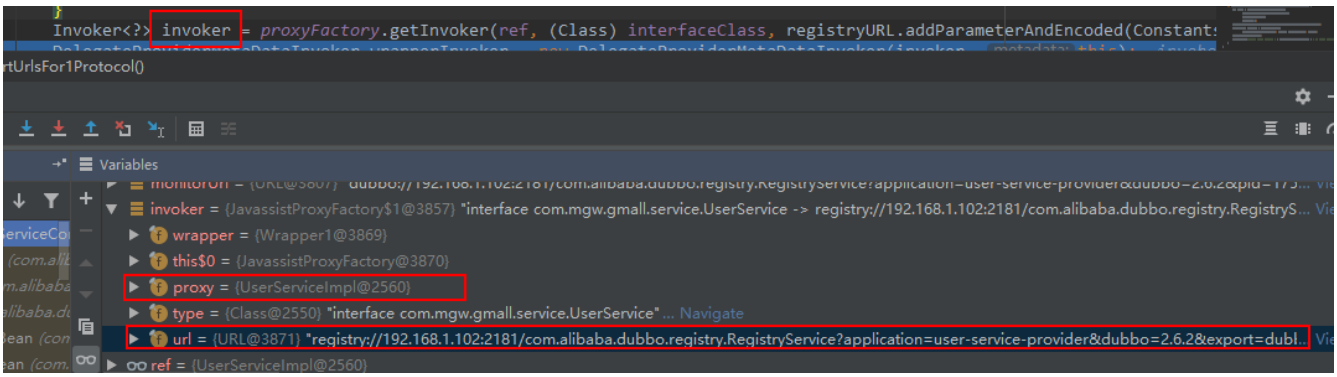
```
                                        if (argclazz.getName().equals(argument.getType())) {
                                            appendParameters(map, argument, method.getName() +
"." + j);
                                            if (argument.getIndex() != -1 &&
argument.getIndex() != j) {
                                                throw new IllegalArgumentException("argument
config error : the index attribute and type attribute not match :index :" + argument.getIndex() +
", type:" + argument.getType());
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    } else if (argument.getIndex() != -1) {
                        appendParameters(map, argument, method.getName() + "." +
argument.getIndex());
                    } else {
                        throw new IllegalArgumentException("argument config must set index or type
attribute.eg: <dubbo:argument index='0' .../> or <dubbo:argument type=xxx .../>");
                    }

                }
            }
        } // end of methods for
    }

    if (ProtocolUtils.isGeneric(generic)) {
        map.put(Constants.GENERIC_KEY, generic);
        map.put(Constants.METHODS_KEY, Constants.ANY_VALUE);
    } else {
        String revision = Version.getVersion(interfaceClass, version);
        if (revision != null && revision.length() > 0) {
            map.put("revision", revision);
        }
        //需要暴露的方法 就是你需要暴露的类中的方法 本示例为:public List<UserAddress>
getUserAddressList(String userId)
        String[] methods = Wrapper.getWrapper(interfaceClass).getMethodNames();
        if (methods.length == 0) {
            logger.warn("NO method found in service interface " + interfaceClass.getName());
            map.put(Constants.METHODS_KEY, Constants.ANY_VALUE);
        } else {
            map.put(Constants.METHODS_KEY, StringUtils.join(new HashSet<String>
(Arrays.asList(methods)), ","));
        }
    }
    if (!ConfigUtils.isEmpty(token)) {
        if (ConfigUtils.isDefault(token)) {
            map.put(Constants.TOKEN_KEY, UUID.randomUUID().toString());
        } else {
            map.put(Constants.TOKEN_KEY, token);
        }
    }
    if (Constants.LOCAL_PROTOCOL.equals(protocolConfig.getName())) {
        protocolConfig.setRegister(false);
        map.put("notify", "false");
```

```
    }
    // export service
    String contextPath = protocolConfig.getContextpath();
    if ((contextPath == null || contextPath.length() == 0) && provider != null) {
        contextPath = provider.getContextpath();
    }
    //拿到主机ip
    String host = this.findConfigedHosts(protocolConfig, registryURLs, map);
    //拿到端口 此例就是20880
    Integer port = this.findConfigedPorts(protocolConfig, name, map);
    URL url = new URL(name, host, port, (contextPath == null || contextPath.length() == 0 ? "" :
contextPath + "/") + path, map);

    if (ExtensionLoader.getExtensionLoader(ConfiguratorFactory.class)
            .hasExtension(url.getProtocol())) {
        url = ExtensionLoader.getExtensionLoader(ConfiguratorFactory.class)
                .getExtension(url.getProtocol()).getConfigurator(url).configure(url);
    }

    String scope = url.getParameter(Constants.SCOPE_KEY);
    // don't export when none is configured
    if (!Constants.SCOPE_NONE.toString().equalsIgnoreCase(scope)) {

        // export to local if the config is not remote (export to remote only when config is
remote)
        if (!Constants.SCOPE_REMOTE.toString().equalsIgnoreCase(scope)) {
            exportLocal(url);
        }
        // export to remote if the config is not local (export to local only when config is local)
        if (!Constants.SCOPE_LOCAL.toString().equalsIgnoreCase(scope)) {
            if (logger.isInfoEnabled()) {
                logger.info("Export dubbo service " + interfaceClass.getName() + " to url " +
url);
            }
            if (registryURLs != null && !registryURLs.isEmpty()) {
                for (URL registryURL : registryURLs) {
                    url = url.addParameterIfAbsent(Constants.DYNAMIC_KEY,
registryURL.getParameter(Constants.DYNAMIC_KEY));
                    //监控中心的url
                    URL monitorUrl = loadMonitor(registryURL);
                    if (monitorUrl != null) {
                        url = url.addParameterAndEncoded(Constants.MONITOR_KEY,
monitorUrl.toFullString());
                    }
                    if (logger.isInfoEnabled()) {
                        logger.info("Register dubbo service " + interfaceClass.getName() + " url "
+ url + " to registry " + registryURL);
                    }
                    //从代理工厂中获取执行器
                    //其实就是把暴露的接口实现方法和url做了包装而已  见下图invoker包装类
                    Invoker<?> invoker = proxyFactory.getInvoker(ref, (Class) interfaceClass,
registryURL.addParameterAndEncoded(Constants.EXPORT_KEY, url.toFullString()));
                    //
                    DelegateProviderMetaDataInvoker wrapperInvoker = new
DelegateProviderMetaDataInvoker(invoker, this);
                    //private static final Protocol protocol =
ExtensionLoader.getExtensionLoader(Protocol.class).getAdaptiveExtension();
```

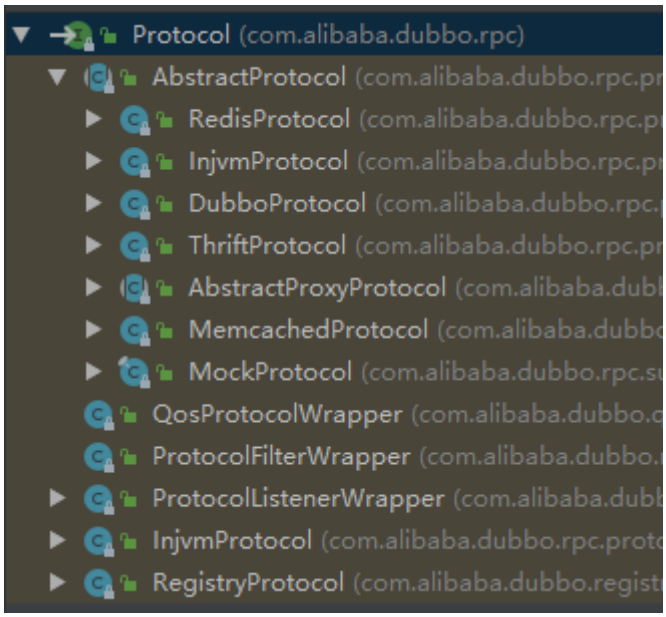
```
                    //使用相应的协议来暴露服务
                    //本例中使用的是dubbo协议来暴露服务
                    Exporter<?> exporter = protocol.export(wrapperInvoker);
                    exporters.add(exporter);
                }
            } else {
                Invoker<?> invoker = proxyFactory.getInvoker(ref, (Class) interfaceClass, url);
                DelegateProviderMetaDataInvoker wrapperInvoker = new
DelegateProviderMetaDataInvoker(invoker, this);

                Exporter<?> exporter = protocol.export(wrapperInvoker);
                exporters.add(exporter);
            }
        }
    }
    this.urls.add(url);
}
//具体的调用过程其实是先调用RegistryProtocol的export方法  然后在RegistryProtocol中再调用到DubboProtocol的
export方法
//RegistryProtocol的export方法
public <T> Exporter<T> export(final Invoker<T> originInvoker) throws RpcException {
    //export invoker
    //做方法的暴露  在这里面调用DubboProtocol的export方法
    final ExporterChangeableWrapper<T> exporter = doLocalExport(originInvoker);

    URL registryUrl = getRegistryUrl(originInvoker);

    //registry provider
    final Registry registry = getRegistry(originInvoker);
    final URL registedProviderUrl = getRegistedProviderUrl(originInvoker);

    //to judge to delay publish whether or not
    boolean register = registedProviderUrl.getParameter("register", true);
    //提供者，消费者的注册表中注册 这步是注册提供者 其实就是个缓存功能  还记得我们说注册中心宕机了其实dubbo也是可以
去调用的，其实就是这里的这个缓存起的作用
    ProviderConsumerRegTable.registerProvider(originInvoker, registryUrl, registedProviderUrl);

    //注册中心注册服务  其实就是利用zookeeper客户端来将相关服务注册到zk上  注册的信息有:这个服务主要是暴露什么接
口，什么类，在哪个服务器上等
    if (register) {
        //注册到zk上
        register(registryUrl, registedProviderUrl);
        ProviderConsumerRegTable.getProviderWrapper(originInvoker).setReg(true);
    }

    // Subscribe the override data
    // FIXME When the provider subscribes, it will affect the scene : a certain JVM exposes the
service and call the same service. Because the subscribed is cached key with the name of the
service, it causes the subscription information to cover.
    final URL overrideSubscribeUrl = getSubscribedOverrideUrl(registedProviderUrl);
    final OverrideListener overrideSubscribeListener = new OverrideListener(overrideSubscribeUrl,
originInvoker);
    overrideListeners.put(overrideSubscribeUrl, overrideSubscribeListener);
    registry.subscribe(overrideSubscribeUrl, overrideSubscribeListener);
    //Ensure that a new exporter instance is returned every time export
    return new DestroyableExporter<T>(exporter, originInvoker, overrideSubscribeUrl,
registedProviderUrl);
```

```
}
//注册服务提供者
public static void registerProvider(Invoker invoker, URL registryUrl, URL providerUrl) {
    /*
     public static ConcurrentHashMap<String, Set<ProviderInvokerWrapper>> providerInvokers = new
ConcurrentHashMap<String, Set<ProviderInvokerWrapper>>();
    public static ConcurrentHashMap<String, Set<ConsumerInvokerWrapper>> consumerInvokers = new
ConcurrentHashMap<String, Set<ConsumerInvokerWrapper>>();
    */
    ProviderInvokerWrapper wrapperInvoker = new ProviderInvokerWrapper(invoker, registryUrl,
providerUrl);
    String serviceUniqueName = providerUrl.getServiceKey();
    Set<ProviderInvokerWrapper> invokers = providerInvokers.get(serviceUniqueName);
    if (invokers == null) {
        //以url为key  以方法执行器为value 注册进一个缓存的map中
        providerInvokers.putIfAbsent(serviceUniqueName, new
ConcurrentHashSet<ProviderInvokerWrapper>());
        invokers = providerInvokers.get(serviceUniqueName);
    }
    invokers.add(wrapperInvoker);
}

//DubboProtocol的export方法
public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
    //拿到地址
    /*
    dubbo://192.168.50.1:20880/com.mgw.gmall.service.UserService?anyhost=true&application=user-
service-
provider&bind.ip=192.168.50.1&bind.port=20880&dubbo=2.6.2&generic=false&interface=com.mgw.gmall.se
rvice.UserService&methods=getUserAddressList&monitor=dubbo%3A%2F%2F192.168.1.102%3A2181%2Fcom.alib
aba.dubbo.registry.RegistryService%3Fapplication%3Duser-service-
provider%26dubbo%3D2.6.2%26pid%3D17524%26protocol%3Dregistry%26refer%3Ddubbo%253D2.6.2%2526interfa
ce%253Dcom.alibaba.dubbo.monitor.MonitorService%2526pid%253D17524%2526timestamp%253D1555765695873%
26registry%3Dzookeeper%26timestamp%3D1555762836973&pid=17524&side=provider&timestamp=1555763482008
    */
    URL url = invoker.getUrl();

    // export service.
    //com.mgw.gmall.service.UserService:20880
    String key = serviceKey(url);
    DubboExporter<T> exporter = new DubboExporter<T>(invoker, key, exporterMap);
    exporterMap.put(key, exporter);

    //export an stub service for dispatching event
    Boolean isStubSupportEvent = url.getParameter(Constants.STUB_EVENT_KEY,
Constants.DEFAULT_STUB_EVENT);
    Boolean isCallbackservice = url.getParameter(Constants.IS_CALLBACK_SERVICE, false);
    if (isStubSupportEvent && !isCallbackservice) {
        String stubServiceMethods = url.getParameter(Constants.STUB_EVENT_METHODS_KEY);
        if (stubServiceMethods == null || stubServiceMethods.length() == 0) {
            if (logger.isWarnEnabled()) {
                logger.warn(new IllegalStateException("consumer [" +
url.getParameter(Constants.INTERFACE_KEY) +
                        "], has set stubproxy support event ,but no stub methods founded."));
            }
        } else {
            stubServiceMethodsMap.put(url.getServiceKey(), stubServiceMethods);
```

```
        }
    }
    //打开服务器  其实就是根据地址，端口创建服务器的过程  更直白的说就是启动netty服务器监听20880
    openServer(url);
    optimizeSerialization(url);
    return exporter;
}
//打开服务器
private void openServer(URL url) {
    // find server.
    String key = url.getAddress();
    //client can export a service which's only for server to invoke
    boolean isServer = url.getParameter(Constants.IS_SERVER_KEY, true);
    if (isServer) {
        //拿到信息交换服务器  一开始是肯定没有的  所以创建
        ExchangeServer server = serverMap.get(key);
        if (server == null) {
            serverMap.put(key, createServer(url));
        } else {
            // server supports reset, use together with override
            server.reset(url);
        }
    }
}
//创建服务器
private ExchangeServer createServer(URL url) {
    // send readonly event when server closes, it's enabled by default
    url = url.addParameterIfAbsent(Constants.CHANNEL_READONLYEVENT_SENT_KEY,
Boolean.TRUE.toString());
    // enable heartbeat by default
    url = url.addParameterIfAbsent(Constants.HEARTBEAT_KEY,
String.valueOf(Constants.DEFAULT_HEARTBEAT));
    String str = url.getParameter(Constants.SERVER_KEY, Constants.DEFAULT_REMOTING_SERVER);

    if (str != null && str.length() > 0 &&
!ExtensionLoader.getExtensionLoader(Transporter.class).hasExtension(str))
        throw new RpcException("Unsupported server type: " + str + ", url: " + url);

    url = url.addParameter(Constants.CODEC_KEY, DubboCodec.NAME);
    ExchangeServer server;
    try {
        //后面不断的绑定  其实最后就是使用netty框架去做的  后面就不再说明了
        server = Exchangers.bind(url, requestHandler);
    } catch (RemotingException e) {
        throw new RpcException("Fail to start server(url: " + url + ") " + e.getMessage(), e);
    }
    str = url.getParameter(Constants.CLIENT_KEY);
    if (str != null && str.length() > 0) {
        Set<String> supportedTypes =
ExtensionLoader.getExtensionLoader(Transporter.class).getSupportedExtensions();
        if (!supportedTypes.contains(str)) {
            throw new RpcException("Unsupported client type: " + str);
        }
    }
    return server;
}
```



注:invoker包装类



注:protocol接口的主要实现的类

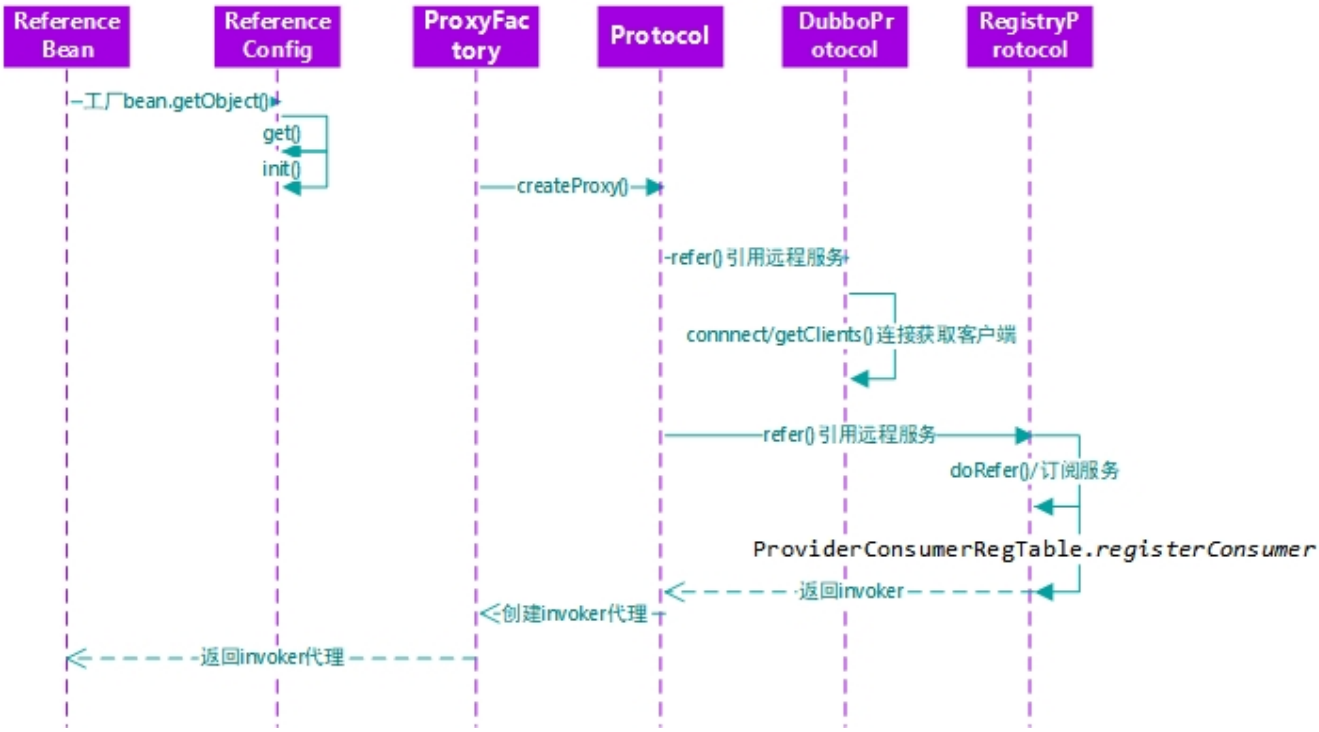
总结服务暴露就两句话:

1.开启netty服务器并监听20880端口

2.注册服务信息到zk上

3.dubbo原理 -服务引用

运行时时序图:



服务引用这里同样用到了初始化配置信息哪里的一个重要的注册类:

```
registerBeanDefinitionParser("reference", new DubboBeanDefinitionParser(ReferenceBean.class,
false));
```

就是这个ReferenceBean的bean

```
/*
<dubbo:reference interface="com.mgw.gmall.service.UserService" id="userService" >
</dubbo:reference>
*/
//这个bean的特殊之处在于他是一个FactoryBean
public class ReferenceBean<T> extends ReferenceConfig<T> implements FactoryBean,
ApplicationContextAware, InitializingBean, DisposableBean {

    private static final long serialVersionUID = 213195494150089726L;

    private transient ApplicationContext applicationContext;

    public ReferenceBean() {
        super();
    }

    public ReferenceBean(Reference reference) {
        super(reference);
    }

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) {
        this.applicationContext = applicationContext;
        SpringExtensionFactory.addApplicationContext(applicationContext);
    }
```

```
    //当FactoryBean调用创建bean时就会调用这个方法
    @Override
    public Object getObject() throws Exception {
        return get();
    }

    @Override
    public Class<?> getObjectType() {
        return getInterfaceClass();
    }

    @Override
    @Parameter(excluded = true)
    public boolean isSingleton() {
        return true;
    }

    @Override
    @SuppressWarnings({"unchecked"})
    public void afterPropertiesSet() throws Exception {
        if (getConsumer() == null) {
            Map<String, ConsumerConfig> consumerConfigMap = applicationContext == null ? null :
BeanFactoryUtils.beansOfTypeIncludingAncestors(applicationContext, ConsumerConfig.class, false,
false);
            if (consumerConfigMap != null && consumerConfigMap.size() > 0) {
                ConsumerConfig consumerConfig = null;
                for (ConsumerConfig config : consumerConfigMap.values()) {
                    if (config.isDefault() == null || config.isDefault().booleanValue()) {
                        if (consumerConfig != null) {
                            throw new IllegalStateException("Duplicate consumer configs: " +
consumerConfig + " and " + config);
                        }
                        consumerConfig = config;
                    }
                }
                if (consumerConfig != null) {
                    setConsumer(consumerConfig);
                }
            }
        }
        if (getApplication() == null
                && (getConsumer() == null || getConsumer().getApplication() == null)) {
            Map<String, ApplicationConfig> applicationConfigMap = applicationContext == null ?
null : BeanFactoryUtils.beansOfTypeIncludingAncestors(applicationContext, ApplicationConfig.class,
false, false);
            if (applicationConfigMap != null && applicationConfigMap.size() > 0) {
                ApplicationConfig applicationConfig = null;
                for (ApplicationConfig config : applicationConfigMap.values()) {
                    if (config.isDefault() == null || config.isDefault().booleanValue()) {
                        if (applicationConfig != null) {
                            throw new IllegalStateException("Duplicate application configs: " +
applicationConfig + " and " + config);
                        }
                        applicationConfig = config;
                    }
                }
                if (applicationConfig != null) {
```

```
                    setApplication(applicationConfig);
                }
            }
        }
        if (getModule() == null
                && (getConsumer() == null || getConsumer().getModule() == null)) {
            Map<String, ModuleConfig> moduleConfigMap = applicationContext == null ? null :
BeanFactoryUtils.beansOfTypeIncludingAncestors(applicationContext, ModuleConfig.class, false,
false);
            if (moduleConfigMap != null && moduleConfigMap.size() > 0) {
                ModuleConfig moduleConfig = null;
                for (ModuleConfig config : moduleConfigMap.values()) {
                    if (config.isDefault() == null || config.isDefault().booleanValue()) {
                        if (moduleConfig != null) {
                            throw new IllegalStateException("Duplicate module configs: " +
moduleConfig + " and " + config);
                        }
                        moduleConfig = config;
                    }
                }
                if (moduleConfig != null) {
                    setModule(moduleConfig);
                }
            }
        }
        if ((getRegistries() == null || getRegistries().isEmpty())
                && (getConsumer() == null || getConsumer().getRegistries() == null ||
getConsumer().getRegistries().isEmpty())
                && (getApplication() == null || getApplication().getRegistries() == null ||
getApplication().getRegistries().isEmpty())) {
            Map<String, RegistryConfig> registryConfigMap = applicationContext == null ? null :
BeanFactoryUtils.beansOfTypeIncludingAncestors(applicationContext, RegistryConfig.class, false,
false);
            if (registryConfigMap != null && registryConfigMap.size() > 0) {
                List<RegistryConfig> registryConfigs = new ArrayList<RegistryConfig>();
                for (RegistryConfig config : registryConfigMap.values()) {
                    if (config.isDefault() == null || config.isDefault().booleanValue()) {
                        registryConfigs.add(config);
                    }
                }
                if (registryConfigs != null && !registryConfigs.isEmpty()) {
                    super.setRegistries(registryConfigs);
                }
            }
        }
        if (getMonitor() == null
                && (getConsumer() == null || getConsumer().getMonitor() == null)
                && (getApplication() == null || getApplication().getMonitor() == null)) {
            Map<String, MonitorConfig> monitorConfigMap = applicationContext == null ? null :
BeanFactoryUtils.beansOfTypeIncludingAncestors(applicationContext, MonitorConfig.class, false,
false);
            if (monitorConfigMap != null && monitorConfigMap.size() > 0) {
                MonitorConfig monitorConfig = null;
                for (MonitorConfig config : monitorConfigMap.values()) {
                    if (config.isDefault() == null || config.isDefault().booleanValue()) {
                        if (monitorConfig != null) {
```

```
                            throw new IllegalStateException("Duplicate monitor configs: " +
monitorConfig + " and " + config);
                        }
                        monitorConfig = config;
                    }
                }
                if (monitorConfig != null) {
                    setMonitor(monitorConfig);
                }
            }
        }
        Boolean b = isInit();
        if (b == null && getConsumer() != null) {
            b = getConsumer().isInit();
        }
        if (b != null && b.booleanValue()) {
            getObject();
        }
    }

    @Override
    public void destroy() {
        // do nothing
    }
}
//通过FatoryBean拿到相关的引用bean
public synchronized T get() {
    if (destroyed) {
        throw new IllegalStateException("Already destroyed!");
    }
    if (ref == null) {
        //进行初始化
        init();
    }
    return ref;
}

private void init() {
    if (initialized) {
        return;
    }
    initialized = true;
    if (interfaceName == null || interfaceName.length() == 0) {
        throw new IllegalStateException("<dubbo:reference interface=\"\" /> interface not allow
null!");
    }
    // get consumer's global configuration
    checkDefault();
    appendProperties(this);
    if (getGeneric() == null && getConsumer() != null) {
        setGeneric(getConsumer().getGeneric());
    }
    if (ProtocolUtils.isGeneric(getGeneric())) {
        interfaceClass = GenericService.class;
    } else {
        try {
            interfaceClass = Class.forName(interfaceName, true, Thread.currentThread()
```

```
                    .getContextClassLoader());
        } catch (ClassNotFoundException e) {
            throw new IllegalStateException(e.getMessage(), e);
        }
        checkInterfaceAndMethods(interfaceClass, methods);
    }
    String resolve = System.getProperty(interfaceName);
    String resolveFile = null;
    if (resolve == null || resolve.length() == 0) {
        resolveFile = System.getProperty("dubbo.resolve.file");
        if (resolveFile == null || resolveFile.length() == 0) {
            File userResolveFile = new File(new File(System.getProperty("user.home")), "dubbo-
resolve.properties");
            if (userResolveFile.exists()) {
                resolveFile = userResolveFile.getAbsolutePath();
            }
        }
        if (resolveFile != null && resolveFile.length() > 0) {
            Properties properties = new Properties();
            FileInputStream fis = null;
            try {
                fis = new FileInputStream(new File(resolveFile));
                properties.load(fis);
            } catch (IOException e) {
                throw new IllegalStateException("Unload " + resolveFile + ", cause: " +
e.getMessage(), e);
            } finally {
                try {
                    if (null != fis) fis.close();
                } catch (IOException e) {
                    logger.warn(e.getMessage(), e);
                }
            }
            resolve = properties.getProperty(interfaceName);
        }
    }
    if (resolve != null && resolve.length() > 0) {
        url = resolve;
        if (logger.isWarnEnabled()) {
            if (resolveFile != null && resolveFile.length() > 0) {
                logger.warn("Using default dubbo resolve file " + resolveFile + " replace " +
interfaceName + "" + resolve + " to p2p invoke remote service.");
            } else {
                logger.warn("Using -D" + interfaceName + "=" + resolve + " to p2p invoke remote
service.");
            }
        }
    }
    if (consumer != null) {
        if (application == null) {
            application = consumer.getApplication();
        }
        if (module == null) {
            module = consumer.getModule();
        }
        if (registries == null) {
            registries = consumer.getRegistries();
```

```
        }
        if (monitor == null) {
            monitor = consumer.getMonitor();
        }
    }
    if (module != null) {
        if (registries == null) {
            registries = module.getRegistries();
        }
        if (monitor == null) {
            monitor = module.getMonitor();
        }
    }
    if (application != null) {
        if (registries == null) {
            registries = application.getRegistries();
        }
        if (monitor == null) {
            monitor = application.getMonitor();
        }
    }
    checkApplication();
    checkStubAndMock(interfaceClass);
    Map<String, String> map = new HashMap<String, String>();
    Map<Object, Object> attributes = new HashMap<Object, Object>();
    map.put(Constants.SIDE_KEY, Constants.CONSUMER_SIDE);
    map.put(Constants.DUBBO_VERSION_KEY, Version.getVersion());
    map.put(Constants.TIMESTAMP_KEY, String.valueOf(System.currentTimeMillis()));
    if (ConfigUtils.getPid() > 0) {
        map.put(Constants.PID_KEY, String.valueOf(ConfigUtils.getPid()));
    }
    if (!isGeneric()) {
        String revision = Version.getVersion(interfaceClass, version);
        if (revision != null && revision.length() > 0) {
            map.put("revision", revision);
        }

        String[] methods = Wrapper.getWrapper(interfaceClass).getMethodNames();
        if (methods.length == 0) {
            logger.warn("NO method found in service interface " + interfaceClass.getName());
            map.put("methods", Constants.ANY_VALUE);
        } else {
            map.put("methods", StringUtils.join(new HashSet<String>(Arrays.asList(methods)),
","));
        }
    }
    map.put(Constants.INTERFACE_KEY, interfaceName);
    appendParameters(map, application);
    appendParameters(map, module);
    appendParameters(map, consumer, Constants.DEFAULT_KEY);
    appendParameters(map, this);
    String prefix = StringUtils.getServiceKey(map);
    if (methods != null && !methods.isEmpty()) {
        for (MethodConfig method : methods) {
            appendParameters(map, method, method.getName());
            String retryKey = method.getName() + ".retry";
            if (map.containsKey(retryKey)) {
```

```
                String retryValue = map.remove(retryKey);
                if ("false".equals(retryValue)) {
                    map.put(method.getName() + ".retries", "0");
                }
            }
            appendAttributes(attributes, method, prefix + "." + method.getName());
            checkAndConvertImplicitConfig(method, map, attributes);
        }
    }

    String hostToRegistry = ConfigUtils.getSystemProperty(Constants.DUBBO_IP_TO_REGISTRY);
    if (hostToRegistry == null || hostToRegistry.length() == 0) {
        hostToRegistry = NetUtils.getLocalHost();
    } else if (isInvalidLocalHost(hostToRegistry)) {
        throw new IllegalArgumentException("Specified invalid registry ip from property:" +
Constants.DUBBO_IP_TO_REGISTRY + ", value:" + hostToRegistry);
    }
    map.put(Constants.REGISTER_IP_KEY, hostToRegistry);

    //attributes are stored by system context.
    StaticContext.getSystemContext().putAll(attributes);
    //上面的其实都是做些信息检查  属性填充等
    //这里其实就是创建个代理类 这里很重要
    ref = createProxy(map);
    ConsumerModel consumerModel = new ConsumerModel(getUniqueServiceName(), this, ref,
interfaceClass.getMethods());
    ApplicationModel.initConsumerModel(getUniqueServiceName(), consumerModel);
}

//创建代理类
private T createProxy(Map<String, String> map) {
    URL tmpUrl = new URL("temp", "localhost", 0, map);
    final boolean isJvmRefer;
    if (isInjvm() == null) {
        if (url != null && url.length() > 0) { // if a url is specified, don't do local reference
            isJvmRefer = false;
        } else if (InjvmProtocol.getInjvmProtocol().isInjvmRefer(tmpUrl)) {
            // by default, reference local service if there is
            isJvmRefer = true;
        } else {
            isJvmRefer = false;
        }
    } else {
        isJvmRefer = isInjvm().booleanValue();
    }

    if (isJvmRefer) {
        URL url = new URL(Constants.LOCAL_PROTOCOL, NetUtils.LOCALHOST, 0,
interfaceClass.getName()).addParameters(map);
        invoker = refprotocol.refer(interfaceClass, url);
        if (logger.isInfoEnabled()) {
            logger.info("Using injvm service " + interfaceClass.getName());
        }
    } else {
        if (url != null && url.length() > 0) { // user specified URL, could be peer-to-peer
address, or register center's address.
            String[] us = Constants.SEMICOLON_SPLIT_PATTERN.split(url);
```

```
            if (us != null && us.length > 0) {
                for (String u : us) {
                    URL url = URL.valueOf(u);
                    if (url.getPath() == null || url.getPath().length() == 0) {
                        url = url.setPath(interfaceName);
                    }
                    if (Constants.REGISTRY_PROTOCOL.equals(url.getProtocol())) {
                        urls.add(url.addParameterAndEncoded(Constants.REFER_KEY,
StringUtils.toQueryString(map)));
                    } else {
                        urls.add(ClusterUtils.mergeUrl(url, map));
                    }
                }
            }
        } else { // assemble URL from register center's configuration
            List<URL> us = loadRegistries(false);
            if (us != null && !us.isEmpty()) {
                for (URL u : us) {
                    URL monitorUrl = loadMonitor(u);
                    if (monitorUrl != null) {
                        map.put(Constants.MONITOR_KEY, URL.encode(monitorUrl.toFullString()));
                    }
                    urls.add(u.addParameterAndEncoded(Constants.REFER_KEY,
StringUtils.toQueryString(map)));
                }
            }
            if (urls == null || urls.isEmpty()) {
                throw new IllegalStateException("No such any registry to reference " +
interfaceName + " on the consumer " + NetUtils.getLocalHost() + " use dubbo version " +
Version.getVersion() + ", please config <dubbo:registry address=\"...\" /> to your spring
config.");
            }
        }

        if (urls.size() == 1) {
            //与服务暴露时一样使用的是protocol 主要涉及两个protocol  一个是RegistryProtocol的refer方法，一个
是DubboProtocol的refer方法
            invoker = refprotocol.refer(interfaceClass, urls.get(0));
        } else {
            List<Invoker<?>> invokers = new ArrayList<Invoker<?>>();
            URL registryURL = null;
            for (URL url : urls) {
                invokers.add(refprotocol.refer(interfaceClass, url));
                if (Constants.REGISTRY_PROTOCOL.equals(url.getProtocol())) {
                    registryURL = url; // use last registry url
                }
            }
            if (registryURL != null) { // registry url is available
                // use AvailableCluster only when register's cluster is available
                URL u = registryURL.addParameter(Constants.CLUSTER_KEY, AvailableCluster.NAME);
                invoker = cluster.join(new StaticDirectory(u, invokers));
            } else { // not a registry url
                invoker = cluster.join(new StaticDirectory(invokers));
            }
        }
    }
```

```
    Boolean c = check;
    if (c == null && consumer != null) {
        c = consumer.isCheck();
    }
    if (c == null) {
        c = true; // default true
    }
    if (c && !invoker.isAvailable()) {
        throw new IllegalStateException("Failed to check the status of the service " +
interfaceName + ". No provider available for the service " + (group == null ? "" : group + "/") +
interfaceName + (version == null ? "" : ":" + version) + " from the url " + invoker.getUrl() + "
to the consumer " + NetUtils.getLocalHost() + " use dubbo version " + Version.getVersion());
    }
    if (logger.isInfoEnabled()) {
        logger.info("Refer dubbo service " + interfaceClass.getName() + " from url " +
invoker.getUrl());
    }
    // create service proxy
    return (T) proxyFactory.getProxy(invoker);
}

//RegistryProtocol的refer方法
public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {
    url = url.setProtocol(url.getParameter(Constants.REGISTRY_KEY,
Constants.DEFAULT_REGISTRY)).removeParameter(Constants.REGISTRY_KEY);
    //根据注册中心地址，获取注册中心的信息
    Registry registry = registryFactory.getRegistry(url);
    if (RegistryService.class.equals(type)) {
        return proxyFactory.getInvoker((T) registry, type, url);
    }

    // group="a,b" or group="*"
    Map<String, String> qs =
StringUtils.parseQueryString(url.getParameterAndDecoded(Constants.REFER_KEY));
    String group = qs.get(Constants.GROUP_KEY);
    if (group != null && group.length() > 0) {
        if ((Constants.COMMA_SPLIT_PATTERN.split(group)).length > 1
                || "*".equals(group)) {
            return doRefer(getMergeableCluster(), registry, type, url);
        }
    }
    //引用  注意传入的参数  注册中心信息    远程引用的接口类型  url
    return doRefer(cluster, registry, type, url);
}

private <T> Invoker<T> doRefer(Cluster cluster, Registry registry, Class<T> type, URL url) {
    RegistryDirectory<T> directory = new RegistryDirectory<T>(type, url);
    directory.setRegistry(registry);
    directory.setProtocol(protocol);
    // all attributes of REFER_KEY
    Map<String, String> parameters = new HashMap<String, String>
(directory.getUrl().getParameters());
    URL subscribeUrl = new URL(Constants.CONSUMER_PROTOCOL,
parameters.remove(Constants.REGISTER_IP_KEY), 0, type.getName(), parameters);
    if (!Constants.ANY_VALUE.equals(url.getServiceInterface())
            && url.getParameter(Constants.REGISTER_KEY, true)) {
```

```
        registry.register(subscribeUrl.addParameters(Constants.CATEGORY_KEY,
Constants.CONSUMERS_CATEGORY,
                Constants.CHECK_KEY, String.valueOf(false)));
    }
    //订阅服务提供者提供的服务  此时会进入到DubboProtocol的refer方法
    directory.subscribe(subscribeUrl.addParameter(Constants.CATEGORY_KEY,
            Constants.PROVIDERS_CATEGORY
                    + "," + Constants.CONFIGURATORS_CATEGORY
                    + "," + Constants.ROUTERS_CATEGORY));

    Invoker invoker = cluster.join(directory);
    //给提供者，消费者注册表中注册消费者的拿到的那些订阅信息
    ProviderConsumerRegTable.registerConsumer(invoker, url, subscribeUrl, directory);
    return invoker;
}

//DubboProtocol的refer方法
public <T> Invoker<T> refer(Class<T> serviceType, URL url) throws RpcException {
    optimizeSerialization(url);
    // create rpc invoker.
    //执行远程引用  核心是获取客户端  其实是已经拿到从注册中心获取到的信息  创建netty客户端进行通信
    DubboInvoker<T> invoker = new DubboInvoker<T>(serviceType, url, getClients(url), invokers);
    invokers.add(invoker);
    return invoker;
}
```

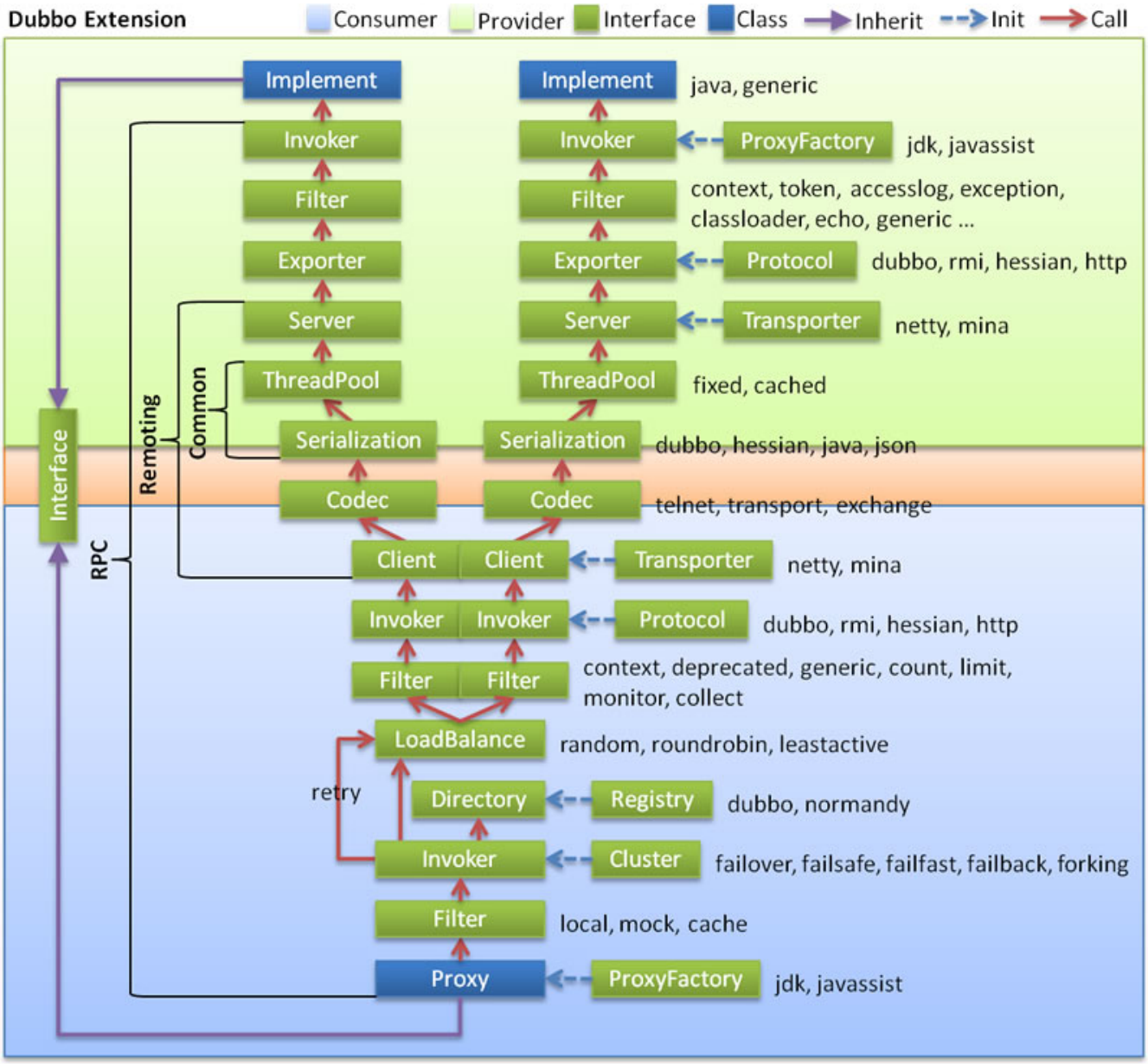
总结起来两句话:

1.创建netty客户端并绑定20880端口通信 订阅出服务

2.订阅出的服务信息，放入本地注册表中

4.服务调用的过程

dubbo官方提供的服务调用链图:

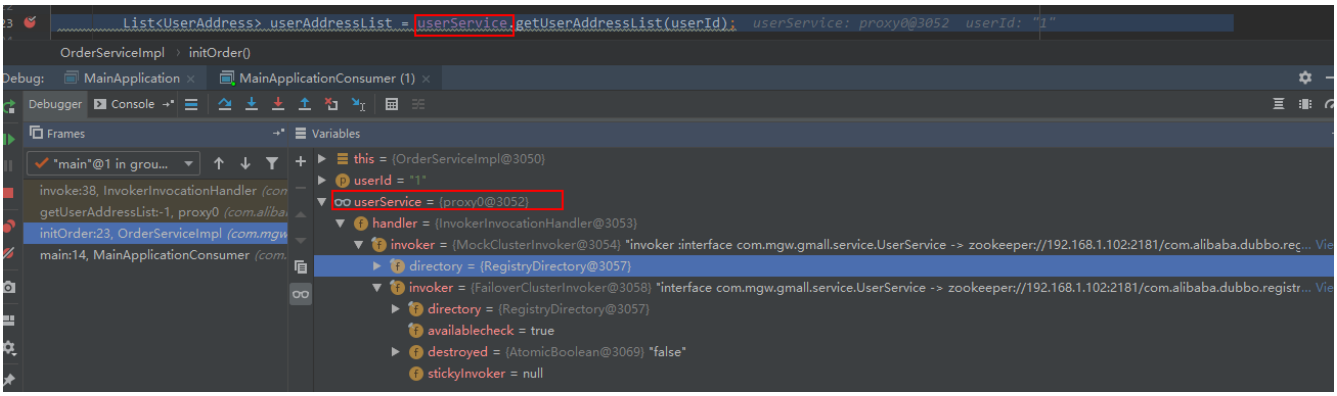


此处执行从从拿到

```
public class OrderServiceImpl implements OrderService {


    //这个自动注入的就是远程rpc的代理对象
    @Autowired
    public UserService userService;

    public void initOrder(String userId) {
        System.out.println("用户ID: "+userId);
        //从此处开始调用  这个userService是代理对象
        List<UserAddress> userAddressList = userService.getUserAddressList(userId);

        for (UserAddress userAddress:userAddressList) {

            System.out.println(userAddress);
        }
    }
```

```
}
```

代理对象部分截图:




```
//dubbo的代理对象的执行器InvokerInvocationHandler
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    String methodName = method.getName();
    Class<?>[] parameterTypes = method.getParameterTypes();
    if (method.getDeclaringClass() == Object.class) {
        return method.invoke(invoker, args);
    }
    if ("toString".equals(methodName) && parameterTypes.length == 0) {
        return invoker.toString();
    }
    if ("hashCode".equals(methodName) && parameterTypes.length == 0) {
        return invoker.hashCode();
    }
    if ("equals".equals(methodName) && parameterTypes.length == 1) {
        return invoker.equals(args[0]);
    }
    //当不是上面的一些常规方法时执行这里的 这里也是层层包装过的
    return invoker.invoke(new RpcInvocation(method, args)).recreate();
}
//调用到MockClusterInvoker的invoke方法
public Result invoke(Invocation invocation) throws RpcException {
    Result result = null;

    String value = directory.getUrl().getMethodParameter(invocation.getMethodName(),
Constants.MOCK_KEY, Boolean.FALSE.toString()).trim();
    if (value.length() == 0 || value.equalsIgnoreCase("false")) {
        //no mock
        //继续调用相关的invoker进行invoke  其实此处的invoker是带有集群容错性质的invoker 叫
FailoverClusterInvoker
        result = this.invoker.invoke(invocation);
    } else if (value.startsWith("force")) {
        if (logger.isWarnEnabled()) {
            logger.info("force-mock: " + invocation.getMethodName() + " force-mock enabled , url :
" + directory.getUrl());
        }
        //force:direct mock
        result = doMockInvoke(invocation, null);
    } else {
        //fail-mock
        try {
```

```
            result = this.invoker.invoke(invocation);
        } catch (RpcException e) {
            if (e.isBiz()) {
                throw e;
            } else {
                if (logger.isWarnEnabled()) {
                    logger.warn("fail-mock: " + invocation.getMethodName() + " fail-mock enabled ,
url : " + directory.getUrl(), e);
                }
                result = doMockInvoke(invocation, e);
            }
        }
    }
    return result;
}

public Result invoke(final Invocation invocation) throws RpcException {
    checkWhetherDestroyed();
    LoadBalance loadbalance = null;
    //从注册中心拿到的相关方法信息  全部封装到一个list中
    List<Invoker<T>> invokers = list(invocation);
    if (invokers != null && !invokers.isEmpty()) {
        //获取带负责均衡机制
        loadbalance =
ExtensionLoader.getExtensionLoader(LoadBalance.class).getExtension(invokers.get(0).getUrl()
                .getMethodParameter(invocation.getMethodName(), Constants.LOADBALANCE_KEY,
Constants.DEFAULT_LOADBALANCE));
    }
    RpcUtils.attachInvocationIdIfAsync(getUrl(), invocation);
    //再次进入invoke中 此时真正进入到FailoverClusterInvoker的invoke中
    return doInvoke(invocation, invokers, loadbalance);
}

//FailoverClusterInvoker的invoke
public Result doInvoke(Invocation invocation, final List<Invoker<T>> invokers, LoadBalance
loadbalance) throws RpcException {
    List<Invoker<T>> copyinvokers = invokers;
    checkInvokers(copyinvokers, invocation);
    int len = getUrl().getMethodParameter(invocation.getMethodName(), Constants.RETRIES_KEY,
Constants.DEFAULT_RETRIES) + 1;
    if (len <= 0) {
        len = 1;
    }
    // retry loop.
    RpcException le = null; // last exception.
    List<Invoker<T>> invoked = new ArrayList<Invoker<T>>(copyinvokers.size()); // invoked
invokers.
    Set<String> providers = new HashSet<String>(len);
    for (int i = 0; i < len; i++) {
        //Reselect before retry to avoid a change of candidate `invokers`.
        //NOTE: if `invokers` changed, then `invoked` also lose accuracy.
        if (i > 0) {
            checkWhetherDestroyed();
            copyinvokers = list(invocation);
            // check again
            checkInvokers(copyinvokers, invocation);
        }
```

```
        //根据负载均衡机制选择一个Invoker
        Invoker<T> invoker = select(loadbalance, invocation, copyinvokers, invoked);
        invoked.add(invoker);
        RpcContext.getContext().setInvokers((List) invoked);
        try {
            //这里开始执行目标方法  但是里面做了各种的包装  需要层层向外解  最终调用到DubboInvoker的doInvoke方
法
            Result result = invoker.invoke(invocation);
            if (le != null && logger.isWarnEnabled()) {
                logger.warn("Although retry the method " + invocation.getMethodName()
                        + " in the service " + getInterface().getName()
                        + " was successful by the provider " + invoker.getUrl().getAddress()
                        + ", but there have been failed providers " + providers
                        + " (" + providers.size() + "/" + copyinvokers.size()
                        + ") from the registry " + directory.getUrl().getAddress()
                        + " on the consumer " + NetUtils.getLocalHost()
                        + " using the dubbo version " + Version.getVersion() + ". Last error is: "
                        + le.getMessage(), le);
            }
            return result;
        } catch (RpcException e) {
            if (e.isBiz()) { // biz exception.
                throw e;
            }
            le = e;
        } catch (Throwable e) {
            le = new RpcException(e.getMessage(), e);
        } finally {
            providers.add(invoker.getUrl().getAddress());
        }
    }
    throw new RpcException(le != null ? le.getCode() : 0, "Failed to invoke the method "
            + invocation.getMethodName() + " in the service " + getInterface().getName()
            + ". Tried " + len + " times of the providers " + providers
            + " (" + providers.size() + "/" + copyinvokers.size()
            + ") from the registry " + directory.getUrl().getAddress()
            + " on the consumer " + NetUtils.getLocalHost() + " using the dubbo version "
            + Version.getVersion() + ". Last error is: "
            + (le != null ? le.getMessage() : ""), le != null && le.getCause() != null ?
le.getCause() : le);
}
//DubboInvoker的doInvoke方法
protected Result doInvoke(final Invocation invocation) throws Throwable {
    RpcInvocation inv = (RpcInvocation) invocation;
    final String methodName = RpcUtils.getMethodName(invocation);
    inv.setAttachment(Constants.PATH_KEY, getUrl().getPath());
    inv.setAttachment(Constants.VERSION_KEY, version);

    //拿到客户端
    ExchangeClient currentClient;
    if (clients.length == 1) {
        currentClient = clients[0];
    } else {
        currentClient = clients[index.getAndIncrement() % clients.length];
    }
    try {
        boolean isAsync = RpcUtils.isAsync(getUrl(), invocation);
```

```
        boolean isOneway = RpcUtils.isOneway(getUrl(), invocation);
        int timeout = getUrl().getMethodParameter(methodName, Constants.TIMEOUT_KEY,
Constants.DEFAULT_TIMEOUT);
        if (isOneway) {
            boolean isSent = getUrl().getMethodParameter(methodName, Constants.SENT_KEY, false);
            currentClient.send(inv, isSent);
            RpcContext.getContext().setFuture(null);
            return new RpcResult();
        } else if (isAsync) {
            //客户端发起请求 模拟出一个请求对象  然后发送出去 得到结果  返回结果
            ResponseFuture future = currentClient.request(inv, timeout);
            RpcContext.getContext().setFuture(new FutureAdapter<Object>(future));
            return new RpcResult();
        } else {
            RpcContext.getContext().setFuture(null);
            return (Result) currentClient.request(inv, timeout).get();
        }
    } catch (TimeoutException e) {
        throw new RpcException(RpcException.TIMEOUT_EXCEPTION, "Invoke remote method timeout.
method: " + invocation.getMethodName() + ", provider: " + getUrl() + ", cause: " + e.getMessage(),
e);
    } catch (RemotingException e) {
        throw new RpcException(RpcException.NETWORK_EXCEPTION, "Failed to invoke remote method: "
+ invocation.getMethodName() + ", provider: " + getUrl() + ", cause: " + e.getMessage(), e);
    }
}
```

总结就一句话:

使用neety客户端，发送请求去服务端获取结构

其实整个dubbo的原理官网给出了非常详细的说明，特在此附上网址，以备诸君查看:http://dubbo.apache.org/zh-cn/docs/dev/design.html